Examining ASP.NET's Membership, Roles, and Profile - Part 18

By Scott Mitchell

Introduction

Membership, in a nutshell, is a framework build into the .NET Framework that supports creating, authenticating, deleting, and modifying user account information. Each user account has a set of core properties: username, password, email, a security question and answer, whether or not the account has been approved, whether or not the user is locked out of the system, and so on. These user-specific properties are certainly helpful, but they're hardly exhaustive - it's not uncommon for an application to need to track additional user-specific properties. For example, an online messageboard site might want to also also associate a signature, homepage URL, and IM address with each user account.

There are two ways to associate additional information with user accounts when using the Membership model. The first - which affords the greatest flexibility, but requires the most upfront effort - is to create a custom data store for this information. If you are using the `SqlMembershipProvider`, this would mean creating an additional database table that had as a primary key the UserId value from the `aspnet_Users` table and columns for each of the additional user properties. The second option is to use the Profile system, which allows additional user-specific properties to be defined in a configuration file. (See Part 6 for an in-depth look at the Profile system.)

This article explores how to store additional user information in a separate database table. We'll see how to allow a signed in user to update these additional user-specific properties and how to create a page to display information about a selected user. What's more, we'll look at using ASP.NET Routing to display user information using an SEO-friendly, human-readable URL like `www.yoursite.com/Users/`*`username`*. Read on to learn more!

Creating a Database Table for Additional User-Specific Properties

When creating an ASP.NET application that supports user accounts and uses the Membership system, it's not uncommon that you need to track user-specific properties in addition to those provided by Membership. Users in the Membership system only have a core set of properties, like username, password, email, security question and answer, last login date, and so on. Membership does not include more general properties like birth date, IM address, or phone number.

Additional user-specific information can be captured in a new database table or through use of ASP.NET's Profile system. The Profile system allows the additional user-specific properties to be defined in the `Web.config` file and is responsible for persisting these values to some data store. How the user-specific properties are serialized and deserialized to a backing store is the responsibility of the configured Profile provider. The default Profile provider - `SqlProfileProvider` - serializes the property values to the `aspnet_Profile` table in a SQL Server database. (See Part 6 for an in-depth look at the Profile system.)

Personally, I'm not a big fan of the `SqlProfileProvider`. As I noted in my Storing Additional User Information Security Tutorial:

> The main benefit of the Profile framework is that it allows for developers to define the profile properties in `Web.config` &#65533; no code needs to be written to serialize the profile data to and from the underlying data store. In short, it is incredibly easy to define a set of profile

properties and to work with them in code. However, the Profile system leaves a lot to be desired when it comes to versioning, so if you have an application where you expect new user-specific properties to be added at a later time, or existing ones to be removed or modified, then the Profile framework may not be the best option. Moreover, the `SqlProfileProvider` stores the profile properties in a highly denormalized fashion, making it next to impossible to run queries directly against the profile data (such as, how many users have a home town of New York).

For these reasons, to store additional user-specific information I prefer creating a new database table named `UserDetails` and adding a column to that table for each needed user-specific property that is not already captured by Membership. Each user who had these additional properties set would have exactly one record in the `UserDetails` table. At this point you may be wondering how we'd associate a record in the `UserDetails` table with a particular user account. Presuming you are using the default provider for Membership (`SqlMembershipProvider`) then your database contains a series of tables that start with the prefix `aspnet_`. The `aspnet_Users` table stores a record for each user account in the system, with each user account uniquely identified by a column named `UserId` of type `uniqueidentifier`. Consequently, to link a record in `UserDetails` back to the Membership user, we need a `UserId` column of type `uniqueidentifier` in `UserDetails` that is both a primary key and a foreign key back to the `aspnet_Users` table's `UserId` column.

Imagine we were building a web application that uses Membership, but needed to track three additional, optional data points for each user: a bio, the user's birth date, and the URL to the user's website. To capture this information we'd create a new database table named `UserDetails` with the following schema:

Column	Data Type	Notes
`UserId`	`uniqueidentifier`	Primary key. Also a foreign key back to `aspnet_Users.UserId`.
`Bio`	`nvarchar(MAX)`	
`BirthDate`	`date`	
`WebsiteUrl`	`nvarchar(256)`	

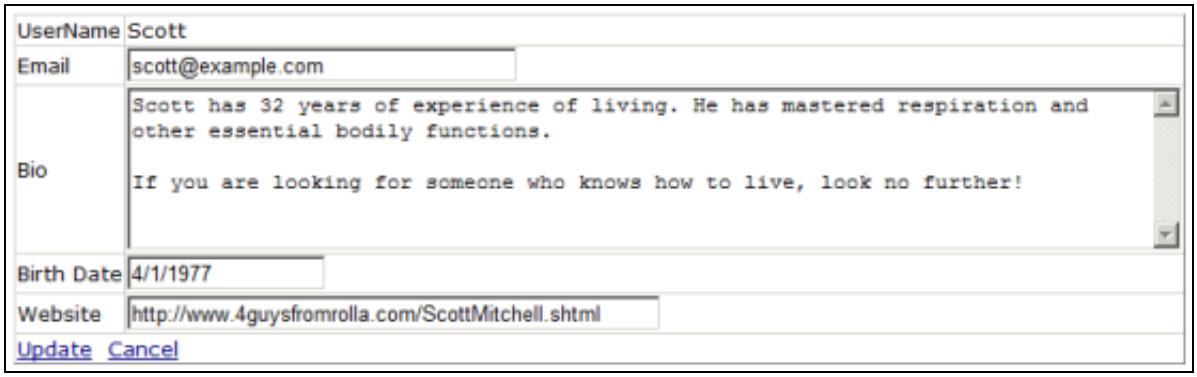
With this table in place, we can now track both the core Membership user information (which is actually stored in two tables - `aspnet_Users` and `aspnet_Membership`) as well as extended user information in `UserDetails`.

Managing User Account Information

ASP.NET offers a number of user account-related Web controls that greatly simplify and expedite building common user account-related web pages. For instance, the Login Web control makes it a cinch to add an interface to collect user credentials; the CreateUserWizard is helpful in creating a registration page. Unfortunately, the Toolbox does not include a Web control for managing user information. Yes, there is the ChangePassword control, but there's no control to let the user change their email address, or to let a user manage any custom properties defined in the Profile system or through a separate table like the `UserDetails` table we just created.

The good news is that creating such an interface isn't terribly difficult. The demo available for download at the end of this article has a page named `~/UsersOnly/ManageProfile.aspx`. (The website is configured such that pages inside the `UsersOnly` folder are only accessible by authenticated users.) The `ManageProfile.aspx` page uses DetailsView and SqlDataSource controls to display the currently logged

on user's username and provides an editing interface that permits the user to change her email address, bio, birth date, and website URL.



Note that there's a zero-to-one relationship between the `aspnet_Users` and `UserDetails` table. All users in the system will have precisely one record in the `aspnet_Users` table, but there's nothing that requires them to also have a record in the UserDetails table. However, if a user does have a record in UserDetails he will have no more than one. In other words, it is possible for a user to exist yet not have a record in `UserDetails`. Any users that existed in the database before adding the `UserDetails` table will not have a record in `UserDetails`, and unless you update the registration process to prompt the user for these additional properties when creating an account, new users won't have a record in `UserDetails`, either.

We must take this into account when populating the data in the DetailsView shown in the screen above. The DetailsView above is populated with data from three tables: the UserName and Email fields come from the `aspnet_Users` and `aspnet_Membership` tables, respectively, while the Bio, Birth Date, and Website fields come `UserDetails`. When pulling this data from the database it is imperative that we use a `LEFT JOIN` on `UserDetails`, as using an `INNER JOIN` would exclude those users that existed in the Membership tables but had no record in `UserDetails`.

Specifically, the DetailsView is populated with the data returned by the following query. The `@UserId` parameter is programmatically set to the `UserId` of the currently logged in user in the ASP.NET page's code-behind class.

```
SELECT aspnet_Users.UserId, Bio, BirthDate, WebsiteUrl, UserName, Email
FROM aspnet_Users
   INNER JOIN aspnet_Membership ON
      aspnet_Users.UserId = aspnet_Membership.UserId
   LEFT OUTER JOIN UserDetails ON
      UserDetails.UserId = aspnet_Users.UserId
WHERE aspnet_Users.UserId = @UserId
```

If a user who does not have any records in the `UserDetails` table visits this page then the database will return a `NULL` value for those columns from the `UserDetails` table - `Bio`, `BirthDate`, and `WebsiteUrl` - thereby leaving those TextBoxes empty.

When the user clicks the DetailsView control's Update Button, we do two things:

1. Add or update the user properties in `UserDetails`, and
2. Update the user's Email address via the Membership API, if needed

Step 1 is handled by a stored procedure named `AddOrUpdateUserDetails` that first checks to see if there exists a record in `UserDetails` whose `UserId` column value matches the UserId of the person visiting the page. If no match is found, a new record is added to the `UserDetails` page. However, if a match if found, that record is updated.

```
CREATE PROCEDURE dbo.AddOrUpdateUserDetails
   (
      @UserId      uniqueidentifier,
      @Bio      nvarchar(max),
      @BirthDate   date,
      @WebsiteUrl   nvarchar(256)
   )
AS
   -- Does this UserId already have a record in UserDetails? If so, UPDATE, else INSERT
   IF EXISTS(SELECT 1 FROM UserDetails WHERE UserId = @UserId)
      UPDATE UserDetails SET
         Bio = @Bio,
         BirthDate = @BirthDate,
         WebsiteUrl = @WebsiteUrl
      WHERE UserId = @UserId

   ELSE
      INSERT INTO UserDetails(UserId, Bio, BirthDate, WebsiteUrl)
      VALUES(@UserId, @Bio, @BirthDate, @WebsiteUrl)
```

Step 2 is handled programmatically in the ASP.NET page's code-behind class in the DetailsView control's `ItemUpdated` event handler. First, the TextBox where the user entered her email address is referenced. Next, details about the currently logged on user are retrieved via a call to `Membership.GetUser`. The email address entered in the TextBox is then compared to the user's `Email` property value; if there is a mismatch then the `Email` property is updated and the user information saved.

```
' Reference the Email TextBox that exists within the DetailsView control's templates
Dim txtEmail As TextBox = CType(dvYourProfile.FindControl("txtEmail"), TextBox)

' Get information about the currently logged on user
Dim usrInfo As MembershipUser = Membership.GetUser()

' If the email address entered does not match the email address on file, then update it!
If String.CompareOrdinal(usrInfo.Email, txtEmail.Text) <> 0 Then
   'They are different!
   usrInfo.Email = txtEmail.Text
   Membership.UpdateUser(usrInfo)
End If
```

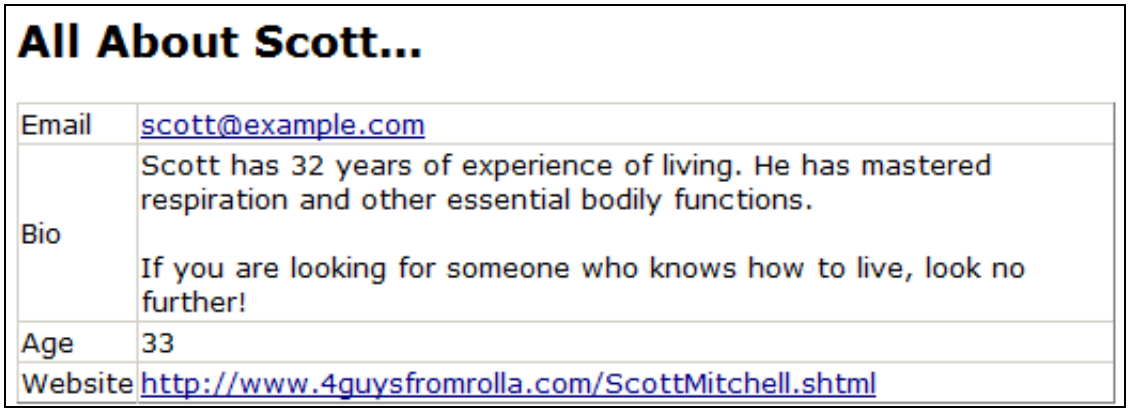
The `ManageProfile.aspx` page allows existing users to log in and add (or update) their information. For some applications, it may be imperative that the user supply this information while initially registering their account. For a look at how to customize the CreateUserWizard control to enable such functionality, refer to my security tutorial, Storing Additional User Information (there's also a C# version).

Displaying User Information To Others

The `~/UsersOnly/ManageProfile.aspx` page allows the currently logged on user to manager her own account, but many sites also provide a page from which anyone can view information about any user. The demo for download at the end of this article includes such a page - `~/UserDetails.aspx`. This `UserDetails.aspx` uses the same database query as the `ManageProfile.aspx` page to retrieve information about the specified user. The key difference is that `ManageProfile.aspx` retrieves information about the currently logged on user whereas `UserDetails.aspx` gets the information about the requested user.

`UserDetails.aspx` displays the results of the query in a DetailsView (like `ManageProfile.aspx`), but is not editable. Moreover, the DetailsView displays the email and website URL using HyperLink controls so that they can be clicked by the person visiting `UserDetails.aspx`. Also, the user's birth date is not shown;

instead, his age is displayed.



At this point you may be wondering how the `UserDetails.aspx` page knows what user's information to get and display. There are a variety of ways we could supply this information to `UserDetails.aspx`. Perhaps the simplest way would be to pass the `UserId` through the querystring, meaning that if you wanted to view details about user Scott (who has a UserId of `3e8195a9-26fa-4efe-9115-66513863c0fc`, you would visit `UserDetails.aspx?UserId=3e8195a9-26fa-4efe-9115-66513863c0fc`. While this certainly works, it leads to some pretty unattractive URLs. Imagine one of our users was having dinner with a colleague and wanted to direct his friend to the page that showed his account information. Can you imagine the awkwardness of saying, "Oh yeah, just visit `www.mysite.com/UserDetails.aspx?UserId=3e8195a9-26fa-4efe-9115-66513863c0fc`!" Chances are, you would not get invited back to dinner!

Ideally the URL for viewing a user's information would be something like `www.yoursite.com/Users/Scott`, or, more generally, `www.yoursite.com/Users/`*`username`*. Such custom URLs are possible with ASP.NET Routing, which is a framework that was first added to ASP.NET version 3.5 SP1. ASP.NET Routing cleanly decouples the web page serving the request from the URL, allowing page developers to create URL patterns that point to an existing web page on the server. Using ASP.NET Routing we can define a URL pattern of the format `Users/`*`username`* that routes to `~/UserDetails.aspx`. From `UserDetails.aspx` we can determine the value passed in the *username* parameter and display the appropriate user information.

As discussed in URL Routing in ASP.NET 4, implementing ASP.NET Routing in an ASP.NET 4 website requires two steps:

1. Add a `Global.asax` file and register the routes of interest, and
2. Create or modify the ASP.NET page referenced by the route so that it examines the route parameters (if necessary)

(If you are using ASP.NET 3.5 SP1, please see Using ASP.NET Routing Without ASP.NET MVC for a list of the steps to implement ASP.NET Routing.)

For this particular example we need only one route, namely the one that maps the URL pattern `Users/`*`username`* to the `~/UserDetails.aspx` page. This is accomplished by the following code in `Global.asax` (see URL Routing in ASP.NET 4 for an example in C#). The call to `MapPageRoute` sets up the routing framework so that any URLs that arrive with the pattern `Users/`*`username`* are routed to `~/UserDetails.aspx`.

```
Sub Application_Start(ByVal sender As Object, ByVal e As EventArgs)
   RegisterRoutes(RouteTable.Routes)
End Sub

Sub RegisterRoutes(ByVal routes As RouteCollection)
   ' Register a route for /Users/{username}
```

```
    routes.MapPageRoute("UserDetailsRoute", "Users/{username}", "~/UserDetails.aspx")
End Sub
```

In `~/UserDetails.aspx` the *username* routing parameter can be accessed in a variety of ways. It can be accessed declaratively, as is done in the Label at the top of the page that displays the user's name whose information is being displayed:

```
<h2>
    All About <asp:Label ID="lblUserName" runat="server" Text="<%$ RouteValue:username %>"
/>...
</h2>
```

The route value can also be accessed declaratively from a data source parameter. `UserDetails.aspx` uses this approach in the SqlDataSource control that populates the DetailsView. Specifically, the database query contains a parameter in its `WHERE` clause - `WHERE aspnet_Users.UserName = @UserName` - and this parameter is defined in the SqlDataSource control's `SelectParameters` collection as a RouteParameter using a `RouteKey` value of `username`. In English (or something closer to it), this means that the value of the `@UserName` parameter in the `SELECT` query is being populated by the *username* route parameter.

Finally, you can retrieve the route parameters programmatically using the `RouteData.Values` collection.

Happy Programming!

- By Scott Mitchell

Further Reading

- Examining ASP.NET's Membership, Roles, and Profile - Part 6
- Accessing and Updating Data in ASP.NET
- Website Security Tutorials (VB and C# versions available)
- Storing Additional User Information Security Tutorial (VB) [C# Version]
- URL Routing in ASP.NET 4.0

Attachments

- Download the code used in this article

Article Information	
Article Title:	ASP.NET.Examining ASP.NET's Membership, Roles, and Profile - Part 18
Article Author:	Scott Mitchell
Published Date:	November 3, 2010
Article URL:	http://www.4GuysFromRolla.com/articles/110310-1.aspx